A Portable MPI Implementation of the SPAI Preconditioner in ISIS++ *

Stephen T. Barnard † Robert L. Clay ‡

Abstract

A parallel MPI implementation of the Sparse Approximate Inverse (SPAI) preconditioner is described. SPAI has proven to be a highly effective preconditioner, and is inherently parallel because it computes columns (or rows) of the preconditioning matrix independently. However, there are several problems that must be addressed for an efficient MPI implementation: load balance, latency hiding, and the need for one-sided communication. The effectiveness, efficiency, and scaling behavior of our implementation will be shown for different platforms.

1 Introduction

The solution of large, sparse systems of linear equations on distributed-memory parallel computers is an important problem in many scientific and engineering disciplines. Since direct solution methods cannot be parallelized effectively, iterative methods such as Conjugate Gradient, GMRES, Bi-CGSTAB, QMR, etc. are typically used [1]. Parallel implementations of these solvers are not difficult to create, but an effective preconditioner is usually required for them converge in a reasonable number of iterations, or even to converge at all. Unfortunately, the widely used ILU-type preconditioners, which are based on incomplete LU factorizations, are impossible (or at least highly problematical) to parallelize. A factored sparse approximate inverse described by Benzi and Tuma [2] produces a preconditioner that can be used in parallel, but the construction of the preconditioner is sequential. On the other hand, the common preconditioners that can be parallelized, such as Polynomial and Block Jacobi, do not seem to be very effective for many important problems. The Sparse Approximate Inverse (SPAI) preconditioner, recently described by Grote and Huckle [5, 6], is an interesting alternative because it is both highly effective and inherently parallel. A theoretical basis for the effectiveness of SPAI has been demonstrated in [5].

Gould and Scott [4] evaluated SPAI on a number of standard examples. Their conclusion was that SPAI is significantly more expensive than ILU-type preconditioners on a single processor, but that SPAI sometimes succeeds when ILU fails. It appears that SPAI should be most useful on multiple processors, where its inherent parallelism would make it reasonably efficient and where ILU would be impractical. However, there are several difficult problems to confront in implementing an efficient, portable parallel implementation

*to appear in: Eighth SIAM Conference for Parallel Processing for Scientific Computing, March 1997

†Research Scientist, MRJ Technology Solutions, NASA Ames Research Center, Moffett Field, CA 94035 (barnard@nas.nasa.gov).

‡Senior Member Technical Staff, Distributed Systems Research Dept., Sandia National Laboratory, Livermore, CA 94550 (rlclay@ca.sandia.gov).

of SPAI. The main problems are "one-sided" communication, load balancing, and latency hiding communication.

A parallel version of SPAI has been implemented in ISIS++, an object-oriented framework for the scalable solution of sparse linear systems of equations[1]. ISIS++ includes a collection of preconditioners and Krylov subspace solution methods, and is designed for portability, scalability, robustness, and adaptivity through run-time component interchangeability. While the ISIS++ framework is capable of addressing generalized solution of sparse linear systems, the focus has been to develop methods suitable for large-scale 3-D finite element modelling applications.

Section 2 presents a brief review of SPAI, closely following Grote and Huckle. In Section 3 we describe the techniques we use to to implement an efficient and portable parallel version of SPAI, which is the main topic of this paper. Section 4 presents performance and scaling results.

2 SPAI

Given a system of linear equations

$$A\mathbf{x} = \mathbf{b} \qquad \mathbf{x}, \mathbf{b} \in \mathbb{R}^n$$

we seek a solution $\mathbf{x} = A^{-1}\mathbf{b}$. An iterative solver starts with an initial guess $\mathbf{x}_0$ and constructs a sequence $\{\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_m\}$ that is intended to converge to acceptable approximation $\mathbf{x}_m$ to $\mathbf{x}$ such that $\|\mathbf{r}_m\|/\|\mathbf{b}\| \leq tol$, where $\mathbf{r}_m = \mathbf{b} - A\mathbf{x}_m$.

Convergence is accelerated by *preconditioning*, in which a matrix M is used as either a right preconditioner,

$$AM\mathbf{y} = \mathbf{b}, \quad \mathbf{x} = M\mathbf{y}$$

or a left preconditioner

$$MA\mathbf{x} = M\mathbf{b}.$$

One should choose M such that $AM \approx I$.

Grote and Huckle [5] describe SPAI as a right preconditioner, but as Gould and Scott [4] point out the SPAI algorithm can also be used to construct a left preconditioner. The choice is determined by the distribution of A: a column-wise distribution leads naturally to a right preconditioner, while a row-wise distribution leads to a left preconditioner. Since the sparse matrix-vector multiplication method in ISIS++ uses a row-wise distribution we have chosen to construct a left preconditioner. To be consistent with [5], however, this section describes SPAI as a right preconditioner.

The basic approach of SPAI is to construct a preconditioning matrix M that approximates A^{-1} in the Frobenius norm:

$$\min_M \|AM - I\|_F^2 = \sum_{k=1}^{n} \min_{\mathbf{m}_k} \|A\mathbf{m}_k - \mathbf{e}_k\|_2^2 \tag{1}$$

where $\mathbf{m}_k$ is the kth column of M. The construction of M is decomposed into the n independent problems of constructing the n columns of M. This is the source of the parallelism of SPAI.

If the sparsity pattern of M is known then the solution of (1) is straightforward, amounting to the solution of n independent least squares problems. Let $\mathcal{J} = \{j \mid \mathbf{m}_k(j) \neq$

[1] URL `http://www.ca.sandia.gov/isis/isis++.html`.

$0\}$ be the set of indices of the nonzero entries of the kth column of M. The set of indices of rows in A that could possibly affect a product with column k is $\mathcal{I} = \{i \mid A(i, \mathcal{J}) \neq 0\}$. To solve (1) we construct the *full* submatrix[2] $\hat{A} = A(\mathcal{I}, \mathcal{J})$, which has $|\mathcal{I}|$ rows and $|\mathcal{J}|$ columns, and solve the problem

$$\min_{\hat{\mathbf{m}}_k} \|\hat{A}\hat{\mathbf{m}}_k - \hat{\mathbf{e}}_k\|_2 \tag{2}$$

where $\hat{\mathbf{e}}_k = \mathbf{e}_k(\mathcal{I})$. This can be done, for example, with a QR decomposition as described in [5].

The main difficulty in constructing an approximate sparse inverse is determining the sparsity pattern of M. Grote and Huckle propose the following method. For each column k of M start with some initial sparsity pattern $\mathcal{J}$, which would typically be diagonal: $\mathcal{J} = \{k\}$. Construct the full submatrix $\hat{A}$ and solve the least squares problem (2) to obtain $\hat{\mathbf{m}}_k$. Let $\mathbf{m}_k(\mathcal{J}) = \hat{\mathbf{m}}_k$, with the residual

$$\mathbf{r} = A(., \mathcal{J})\hat{\mathbf{m}}_k - \mathbf{e}_k \ . \tag{3}$$

Assuming that $\|\mathbf{r}\|_2 \neq 0$, then $\mathbf{m}_k$ is not exactly the kth column of the true inverse, and we must *augment* the sparsity structure $\mathcal{J}$ to obtain a better approximation, so we look at how to reduce the magnitude of the nonzero components of the residual.

Let $\mathcal{L} = \{l \mid \mathbf{r}(l) \neq 0\}$. Let $\tilde{\mathcal{J}} = \{j \mid A(\mathcal{L}, j) \neq 0\} \backslash \mathcal{J}$. These are candidate indices to add to $\mathcal{J}$, but there may be very many of them, so it is necessary to somehow choose the ones that most effectively reduce $\|\mathbf{r}\|_2$. Grote and Huckle suggest as a heuristic solving a one-dimensional minimization problem for each $j \in \tilde{\mathcal{J}}$:

$$\min_{\mu_j} \|\mathbf{r} + \mu_j A\mathbf{e}_j\|_2 \tag{4}$$

which has the solution

$$\mu_j = \frac{\mathbf{r}^T A\mathbf{e_j}}{\|A\mathbf{e}_j\|_2^2} \tag{5}$$

with the residual

$$\rho_j = \|\mathbf{r}\|_2^2 - \frac{(\mathbf{r}^T A\mathbf{e_j})^2}{\|A\mathbf{e}_j\|_2^2} \tag{6}$$

The procedure for choosing new indices to augment the sparsity structure $\mathcal{J}$ is as follows:

1. Determine $\tilde{\mathcal{J}}$,
2. Determine ρ_j for all $j \in \tilde{\mathcal{J}}$,
3. Determine the mean of $\{\rho_j\}$,
4. Retain all indices in $\tilde{\mathcal{J}}$ corresponding to a value of ρ less than or equal to the mean, up to to some maximum number of indices (typically 5).

The complete SPAI algorithm is shown in Figure 1. Note that the accuracy of the sparse-inverse approximation is determined by the parameter ϵ.

[2] Note that we store and operate on $\hat{A}$ as a dense matrix, although it may contain zero entries.

```
1.    for all k ∈ {1,...,n} do        // construct a column of M
2.        J ← {k}                     // initial sparsity pattern is diagonal
3.        I ← {i | A(i,J) ≠ 0}
4.        r_norm ← ∞
5.        s ← 1
6.        while (r_norm > ε and s < s_max) do
7.            s ← s + 1
8.            Â ← full_matrix(A,I,J)
9.            solve for m̂ in eq. (2)
10.           r ← A(.,J)m̂ − e_k          // residual
11.           r_norm ← ||r||_2
12.           if r_norm > ε then         // augment sparsity and try again
13.               L ← {l | r(l) ≠ 0}
14.               J̃ ← {j | A(L,j) ≠ 0}\J
15.               for all j ∈ J̃ do
16.                   ρ_j ← minimize(A,r,j)   // using eq. (3)
17.               end for all
18.               J̃' ← best(J̃,r,{ρ_j})
19.               I ← I ⋃ {i | A(i,J̃') ≠ 0}
20.               J ← J ⋃ J̃'
21.           end if
22.       end while
23.       M(.,k) ← J
24.   end for all
```

FIG. 1. *The SPAI Algorithm.*

3 Implementation

Although SPAI is an inherently parallel algorithm, there are several difficult issues to confront in creating an efficient and portable implementation. Deshpande, et al. [3] describe a parallel implementation of a variant of SPAI that relies on the matrix being "structurally symmetric" and which exploits matrix partitioning to reduce interprocessor communication. We also allow for partitioning, but our code implements exactly the algorithm described in [5, 6].

3.1 One-Sided Communication

SPAI computes every row of M independently, but to do so it must access potentially any row of A in a completely unpredictable way. A processor that computes a row of M must therefore access rows of A that reside on other processors. This is straightforward on a shared-memory architecture, but on a distributed-memory system with no support for shared-memory programming it requires either expensive and nonscalable all-to-all communication or so-called "one-sided" communication. We use MPI for maximum portability, but MPI does not support one-sided communication directly. It does, however, provide the functionality to implement one-sided communication in a specialized way.

The processors computing rows of M run entirely asynchronously, with no barriers until M is completed. Whenever a processor needs access to data on another processor, or when

it needs to inform another processor of some condition, it sends a request to that processor in the form of a short message. These requests are handled by a *communications server* that uses the `MPI_Iprobe` function to detect the arrival of requests.

There are five types of requests, distinguished by their message tags in the communications server:

1. Another processor needs a row of A.

2. Another processor needs a row of M. This is part of the load balance mechanism described below.

3. Another processor is storing a row of M. Again, this is part of the load balancing mechanism.

4. A processor has finished constructing all the rows of M that it "owns" and is informing the master processor that it has finished its local work (although it may still construct rows owned by other processors until all processors have finished their local work).

5. The master processor informs all other processors that the construction of M has been completed.

The communications server is called periodically by every processor, typically when they are waiting for remote data or when they have finished a substantial amount of work, such as computing a row of M.

3.2 Latency Hiding

Many distributed-memory computers have large latency in interprocessor communication. The parallel SPAI code masks this latency as much as possible by using asynchronous communication and overlapping work with communication. For example, when a processor initiates a request for a row of A to another processor it uses the asynchronous `MPI_Isend` function, then it repeatedly calls the communications server to service requests from other processors until the data that it requested arrives.

One effective way that the parallel SPAI code hides latency is to avoid unnecessary communication altogether by caching remote references. When a processor is working on a row of M and needs to retrieve a row of A from another processor it puts that row in a cache (implemented with a hash table). It is very likely that subsequent columns of M will require the same row of A, which they will find in the cache without resorting to unnecessary communication. The function that accesses rows of A works as follows:

1. If the row is local simply return it.

2. Otherwise, if it is in the cache return it.

3. Otherwise, initiate a request to the processor that owns it.

4. Service requests until the data arrives and the request queue is empty.

5. Put the row in the cache and return it.

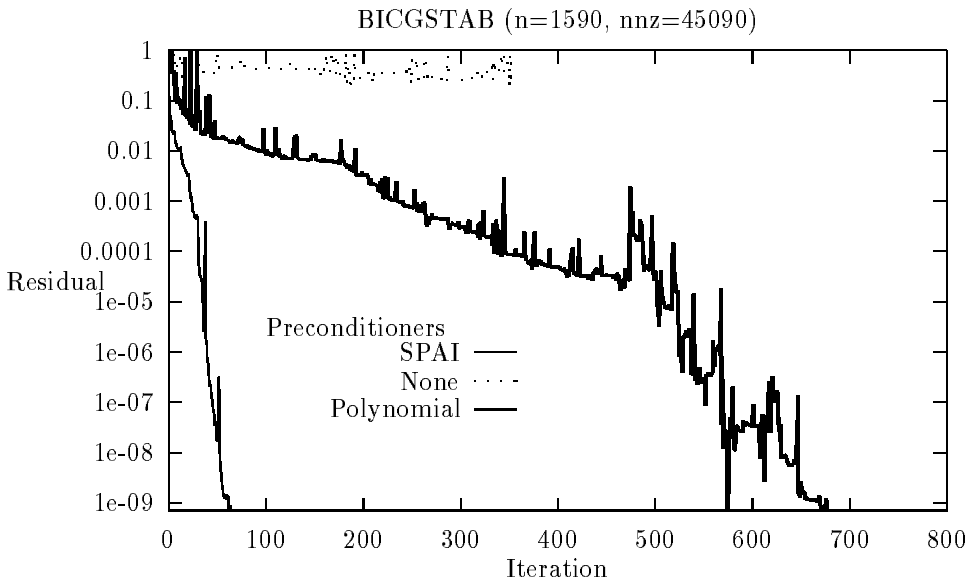

FIG. 2. *Convergence of Bi-CGSTAB.* The Convergence of Bi-CGSTAB applied to a small FEM matrix is shown, using the SPAI preconditioner, no preconditioning, and a polynomial preconditioner.

3.3 Load Balancing

It is very likely that some rows of M will require much more work than the average row, which can lead to a serious load imbalance. Furthermore, it is impossible to predict accurately how much work a row will require, and therefore it is impossible to allocate work to processors ahead of time in a load-balanced distribution. We have implemented a dynamic load balancing strategy to deal with this problem.

Every processor "owns" a number of rows of the matrices A and M, which are assigned at the outset of the program. The indices of the "local" rows of M are maintained as a queue and each processor constructs its local part of M by taking indices from the queue. Suppose processor p reaches the end of the queue, having completed its local work. It sends a message informing the master processor that it has finished its local work, but there may be other processors which are not finished, so processor p polls the other processors, using the communications server, asking whether they have any row indices of M remaining in their queues. Suppose processor q has such an index. It takes that index from the queue and returns it to processor p, which then computes the row of M in exactly the same way as it would compute a local row of M, and when it is finished it returns the row to processor p to be stored in the proper place. When the master processor detects that all processors have finished their local work it sends messages (which are handled by the communications server) to the other processors informing them that M is complete.

4 Performance

We have found SPAI to be very efficient for the FEM problems we have investigated when we consider the complete time-to-solution, including the time required for the solver. While SPAI is much more costly than other parallel preconditioners, it is so much more effective in improving convergence that it is well worth the cost. Figure 2 illustrates the convergence of

# processors	T3D (sec)	SP2 (sec)
1		619.45
2		355.83
4		217.51
8	86.24	121.16
16	49.75	64.51
32	30.44	35.76
64	22.39	23.12
128	13.38	12.87

TABLE 1

Time to Compute SPAI. This table shows the time required to construct the SPAI preconditioner (using $\epsilon = .4$) on various numbers of processors of the Cray T3D and the IBM SP2. The matrix A is of order 49600 and has 3153942 nonzero coefficients.

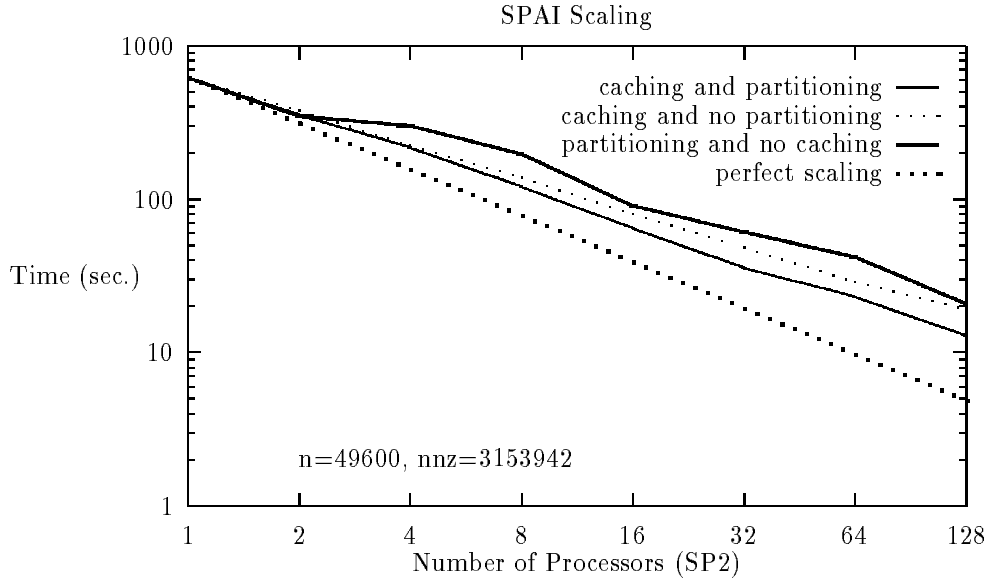

FIG. 3. *SPAI Scaling (SP2).* The scaling of SPAI on the IBM SP2 is shown under three conditions: caching remote references and partitioning the matrix A to minimize communication, no caching of remote references, and no matrix partitioning.

Bi-CGSTAB on a small but otherwise typical FEM problem using SPAI. Extensive analysis of the effectiveness of SPAI can be found in [4].

Table 1 shows the run time for constructing the SPAI preconditioner for a much larger matrix on several different configurations of T3D and SP2 processors. Note that the T3D required eight processors to hold this matrix.

Figure 3 shows how the scaling of SPAI on the SP2 is affected by the techniques of caching remote references and partitioning described in Section 3. In this example both caching and partitioning each increase the performance of 128 processors by about 50-60 percent.

5 Conclusions

The SPAI preconditioner has proven to be an effective tool for certain types of problems. It is also efficent compared to the alternative parallel preconditioners when we consider time-to-solution (including the solver time). Spending more effort to construct a very effective SPAI preconditioner, thereby greatly accelerating the convergence of the solver, is ultimately more efficient than using a very cheap but relatively ineffective parallel preconditioner. This implementation is most appropriate for very large problems that use unstructured meshes and are solved on distributed-memory parallel computers. In these cases serial preconditioners such as ILU are not even feasible, either because the problems cannot be fit into one processor's memory or because the preconditioner would present a serial bottleneck.

The one-sided communication method embodied in the communication server permits random access to remote data with acceptable efficiency. The communication server also permits the code to achieve nearly perfect load balance. It is possible that this technique may prove very useful for a wide variety unstructured applications. One of the authors (Barnard) is currently investigating using a variant of the communications server for a particle-in-cell technique in a parallel molecular dynamics code.

References

[1] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst, *TEMPLATES for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM Publications, 1994.

[2] M. Benzi and M. Tuma, *A sparse approximate inverse preconditioner for nonsymmetric linear systems*, SIAM J. Scientific. Computing, In press.

[3] V. Deshpande, M. J. Grote, P. Messmer, and W. Sawyer, *Parallel implementation of a sparse approximate inverse preconditioner*, Proceeding of Irregular '96 (Santa Barbara).

[4] N. I. M. Gould and J. A. Scott, *On approximate-inverse preconditioners*, RAL 95-026, Computing and Information Systems Department, Atlas Centre, Rutherford Appleton Laboratory, Oxfordshire, England, June 23, 1995.

[5] M. J. Grote and T. Huckle, *Parallel preconditioning with sparse approximate inverses*, SIAM J. Scientific Computing, In press.

[6] M. J. Grote and T. Huckle, *Effective parallel preconditioning with sparse approximate inverses*, In *Proc. SIAM Conf. on Parallel Processing for Scientific Comp.*, San Francisco, pp. 466–471. SIAM, 1995